Contents

• **TCP in NS-3**
  • Introduction of TCP
  • TCP & TCP option in ns-3
  • Practice 1 : Implementation of Autotuning using window scale option

• **MPTCP in NS-3**
  • What is Multi-Path TCP (MPTCP)?
  • TCP extension for MPTCP in ns-3
  • Practice 2 : Comparison of congestion control algorithm of MPTCP
  • Practice 3 : Implementation of hierarchical receive buffer
**Transmission Control Protocol (TCP)**

- **TCP is the most widely used transport protocol**

- **Connection-oriented**
  - Three way handshake to establish the connection

- **Reliable delivery**
  - Retransmission, in-order delivery, and no duplicate

- **Byte stream transmission**
  - Data is divided into segments
## TCP Header Format

<table>
<thead>
<tr>
<th>Source port address (16 bits)</th>
<th>Destination port address (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number (32 bits)</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment number (32 bits)</td>
<td></td>
</tr>
<tr>
<td>Header Length (4 bits)</td>
<td>Reserved (4 bits)</td>
</tr>
<tr>
<td>Flags (8 bits)</td>
<td>Window (16 bits)</td>
</tr>
<tr>
<td>Checksum (16 bits)</td>
<td>Urgent pointer (16 bits)</td>
</tr>
<tr>
<td>Options and Padding (0-320 bits)</td>
<td></td>
</tr>
</tbody>
</table>

- **Length**: 20~60 Bytes, depending on options
- **TCP options**
  - **Window scale**: increase the maximum window field from 64 KBytes to 1 GBytes (RFC 1323)
  - **Selective ACK**: acknowledge discontinuous blocks of packets (RFC 2018)
  - **Timestamp**: for more accurate RTT estimation (RFC 1323)
  - Above 3 options are widely used with commercial applications (e.g., Facebook, Dropbox, Youtube, etc.)
TCP Congestion Control

• A core of the Internet protocol suite (TCP/IP)

• TCP Congestion Control:
  • Achieve high network performance and avoid congestion control

• A mixture of several algos.
  • Congestion Avoidance
  • Slow Start
  • Timeout (re-clocking).
TCP Congestion Control

- **Self-clocking system**
  - Sliding window based flow control
  - Tends to be very stable under a wide range of network environment

http://www.soi.wide.ad.jp/class/20020032/
Congestion Avoidance

- Should share the bottleneck link fairly

- AIMD (Additive Increase Mutiplicative Decrease)
  - Increase TX rate linearly with no event
  - Decrease TX rate to a half on a packet loss

http://blog.leith.ie/some-ways-to-break-tcp/
**Slow Start**

- **Provide high performance, quickly**
  - At the beginning, TCP has no information about the net.
  - It may take long to use the capacity in full

- **Slow start provides a “quick” start**
  - by increasing the TX rate exponentially at the beginning

“Unknown” bottleneck link capacity
Retransmission TimeOut (RTO)

- A kind of back-up plan when an unexpected event occurs
  - E.g., route changes due to power-off of intermediate node
  - Restart congestion control from the beginning
  - Re-establish the self-clocking
  - Commonly, set to several round-trip times (RTTs).
Fast Retransmit

- RTO often takes too long
- Fast Retransmit:
  - At the sender, a packet is considered as being lost, based on the feedback from the receiver (3 duplicate ACKs)
Window Evolution


http://www.soi.wide.ad.jp/class/20020032/
Fast Recovery

- **Fast Recovery**
  - Recover the expected packet loss (single packet loss) “without Slow Start”
  - More efficient operation
- **TCP Tahoe + Fast Recovery = TCP Reno (1990)**

[Graph showing Fast Recovery process]
TCP Congestion Control (Summary)

- A mixture of several algorithms
  - Congestion Avoidance
  - Slow Start
  - Timeout (re-clocking)
  - Fast Retransmit and Fast Recovery

http://monet.postech.ac.kr/research.html
The TCP Sender Does....

- **Measure round-trip time (RTT)**
  - Both mean and variance
  - Used to set RTO timer value
  - Accurate measurement with timestamp option

- **Control congestion window (Congwin, or w)**
  - Control transmission rate by limiting the maximum number of packets in-flight (i.e., sent but not-acknowledged)

\[
\text{throughput} = \frac{w \times MSS}{RTT} \quad \text{Bytes/sec}
\]

MSS: Maximum Segment Size
Pseudo Code of Congestion Avoidance

/* slowstart is over */
* Congwin > threshold */

Until (loss event) {
    every w segments ACKed:
    Congwin++
}

threshold = Congwin/2
Congwin = 1
perform slowstart

1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs
TCP implementation in NS-3 (ver. 3.10~)

- Inheritance structure

```
Socket
  ↓
TcpSocket
  ↓
TcpSocketBase
  ↓
TcpReno
  ↓
TcpNewReno
  ↓
TcpTahoe
```

```
TcpHeader
  ↓
TcpRxBuffer
  ↓
TcpTxBuffer
```

- `src/internet/model/tcp-socket-base.cc/h`
  - Implementation of common TCP operation
  - 3-way handshake, send/receive packet, ACK/Dupack, RTO ....
TcSocket Class Attribute

- SndBufSize : TcSocket maximum transmit buffer size (bytes)
- RcvBufSize : TcSocket maximum receive buffer size (bytes)
- SegmentSize : TCP maximum segment size in bytes
- SlowStartThreshold : TCP slow start threshold (bytes)
- InitialCwnd : TCP initial congestion window size (segments)
- ConnTimeout : TCP retransmission timeout when opening connection (seconds)
- ConnCount : Number of connection attempts (SYN retransmissions) before returning failure
- DelAckTimeout : Timeout value for TCP delayed acks, in seconds
- DelAckCount : Number of packets to wait before sending a TCP ack
- TcpNoDelay : Set to true to disable Nagle's algorithm
- PersistTimeout : Persist timeout to probe for rx window
TcpSocketBase Class Attribute and Trace

**Attribute**

- **MaxSegLifetime** : Maximum segment lifetime in seconds, used for TIME_WAIT state transition to CLOSED state
- **MaxWindowSize** : Max size of advertised window
- **IcmpCallback** : Callback invoked whenever an icmp error is received on this socket
- **IcmpCallback6** : Callback invoked whenever an icmpv6 error is received on this socket

**Trace**

- **RTO** : Retransmission timeout
- **RTT** : Last RTT sample
- **NextTxSequence** : Next sequence number to send (SND.NXT)
- **HighestSequence** : Highest sequence number ever sent in socket's life time
- **State** : TCP state
- **RWND** : Remote side's flow control window
TCP Header Format (Revisit)

<table>
<thead>
<tr>
<th>Source port address (16 bits)</th>
<th>Destination port address (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number (32 bits)</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment number (32 bits)</td>
<td></td>
</tr>
<tr>
<td>Header Length (4 bits)</td>
<td>Reserved (4 bits)</td>
</tr>
<tr>
<td>Flags (8 bits)</td>
<td>Window (16 bits)</td>
</tr>
<tr>
<td>Checksum (16 bits)</td>
<td>Urgent pointer (16 bits)</td>
</tr>
<tr>
<td>Options and Padding ((0-320 bits))</td>
<td></td>
</tr>
</tbody>
</table>

- **Length**: 20~60 Bytes, depending on options
- **TCP options**
  - **Window scale**: increase the maximum window field from 64 KBytes to 1 GBytes (RFC 1323)
  - **Selective ACK**: acknowledge discontinuous blocks of packets (RFC 2018)
  - **Timestamp**: for more accurate RTT estimation (RFC 1323)
  - Above 3 options are widely used with commercial applications (e.g., Facebook, Dropbox, Youtube, etc.)
  - **MPTCP**: TCP extension for Multipath on multi-address (RFC 6824)
Another Window Control

- TCP Flow control
  - The receiver may not have enough buffer space
  - “window” field in the header is written by the receiver to let the sender know the remaining buffer space
  - We often denote this information as “receive window (rwnd)”: the amount of available buffer space at the receiver
  - The sender’s actual transmission window = \(\min(cwnd, rwnd)\)
Procedure of packet sending of TCP in ns3 (1)

Application

1. Send () : check the state of TCP

STM of TCP

States_t=ESTABLISHED

2. SendPendingData() : Check the window size and tx buffer

3. SendDataPacket() : Add TCP header and Option

4. SendPacket() : tx packet to l3layer

Scenario file

Transport layer

TcpSocketBase

IP layer

TcpL4Protocol

States_t=ESTABLISHED

Tx_buffer
Call Send() in Application

• Application for file transfer (example/tcp/tcp-large-transfer.cc)

```c++
void WriteUntilBufferFull (Ptr<Socket> localSocket, uint32_t txSpace)
{
    while (currentTxBytes < totalTxBytes && localSocket->GetTxAvailable () > 0)
    {
        uint32_t left = totalTxBytes - currentTxBytes;
        uint32_t dataOffset = currentTxBytes % writeSize;
        uint32_t toWrite = writeSize - dataOffset;
        toWrite = std::min (toWrite, left);
        toWrite = std::min (toWrite, localSocket->GetTxAvailable ());
        int amountSent = localSocket->Send (&data[dataOffset], toWrite, 0);
        if (amountSent < 0)
        {
            // we will be called again when new tx space becomes available.
            return;
        }
        currentTxBytes += amountSent;
    }
    localSocket->Close ();
```
Send( ) → SendPendingDate()

- `scr/internet/model/tcp-socket-base.cc`

check the state of TCP
SendPendingDate() → sendDataPacket()

```cpp
bool
TcpSocketBase::SendPendingData (bool withAck)
{
    NS_LOG_FUNCTION (this << withAck);
    if (m_txBuffer.Size () == 0)
    {
        return false; // Nothing to send
    }
    if (m_endPoint == 0 && m_endPoint6 == 0)
    {
        NS_LOG_INFO ("TcpSocketBase::SendPendingData: No endpoint; m_shutdownSend=" << m_shutdownSend);
        return false; // Is this the right way to handle this condition?
    }
    uint32_t nPacketsSent = 0;
    while (m_txBuffer.SizeFromSequence (m_nextTxSequence))
    {
        uint32_t w = AvailableWindow (); // Get available window size
        
        uint32_t s = std::min (w, m_segmentSize); // Send no more than window
        uint32_t sz = sendDataPacket (m_nextTxSequence, s, withAck);
        nPacketsSent++;
        // Count sent this loop
        m_nextTxSequence += sz;
        // Advance next tx sequence
    }
    NS_LOG_LOGIC ("SendPendingData sent " << nPacketsSent << " packets");
    return (nPacketsSent > 0);
}
```
SendDataPacket() \rightarrow \text{SendPacket()}

```
01935: uint32_t
01936: TcpSocketBase::SendDataPacket (SequenceNumber32 seq, uint32_t maxSize, bool withAck)
01937: {
01938:     NS_LOG_FUNCTION (this << seq << maxSize << withAck);
01939:

02007:     TcpHeader header;
02008:     header.SetFlags (flags);
02009:     header.SetSequenceNumber (seq);
02010:     header.SetAckNumber (m_rxBuffer->NextRxSequence ());
02011:     if (m_endPoint)
02012:     {
02013:         header.SetSourcePort (m_endPoint->GetLocalPort ());
02014:         header.SetDestinationPort (m_endPoint->GetPeerPort ());
02015:     }
02016:     else
02017:     {
02018:         header.SetSourcePort (m_endPoint6->GetLocalPort ());
02019:         header.SetDestinationPort (m_endPoint6->GetPeerPort ());
02020:     }
02021:     header.SetWindowSize (AdvertisedWindowSize ());
02022:     AddOptions (header);
02023:
02038:     if (m_endPoint)
02039:     {
02040:         m_tcp->SendPacket (p, header, m_endPoint->GetLocalAddress (),
02041:             m_endPoint->GetPeerAddress (), m_boundnetdevice);
02042:     }
```

Add TCP header and Option
Determine and obtain the congestion window size (1)

- Then, how to determine the window size??
- examples/tcp/tcp-variants-comparison.cc

```cpp
// Select TCP variant
if (transport_prot.compare("TcpTahoe") == 0)
    Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue (TcpTahoe::GetTypeId()));
else if (transport_prot.compare("TcpReno") == 0)
    Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue (TcpReno::GetTypeId()));
else if (transport_prot.compare("TcpNewReno") == 0)
    Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue (TcpNewReno::GetTypeId()));
else if (transport_prot.compare("TcpWestwood") == 0)
    {
    // the default protocol type in ns3::TcpWestwood is WESTWOOD
    Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue (TcpWestwood::GetTypeId()));
    Config::SetDefault("ns3::TcpWestwood::FilterType", EnumValue (TcpWestwood::TUSTIN));
    }
else if (transport_prot.compare("TcpWestwoodPlus") == 0)
    {
    Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue (TcpWestwood::GetTypeId()));
    Config::SetDefault("ns3::TcpWestwood::ProtocolType", EnumValue (TcpWestwood::WESTWOODPLUS));
    Config::SetDefault("ns3::TcpWestwood::FilterType", EnumValue (TcpWestwood::TUSTIN));
    }
else
    {
    NS_LOG_DEBUG ("Invalid TCP version");
    exit (1);
    }
```

Set SocketType in TcpL4Protocol
Determine and obtain the congestion window size (2)

- `scr/internet/model/tcp-l4-protocol.cc`

```cpp
0065:   static TypeId tid = TypeId ("ns3::TcpL4Protocol")
0066:   .SetParent<IpL4Protocol> ()
0067:   .AddConstructor<TcpL4Protocol> ()
0068:   .AddAttribute ("RttEstimatorType",
0069:     "Type of RttEstimator objects.",
0070:     RttMeanDeviation::GetTypeId ()),
0071:     MakeTypeIdAccessor (&TcpL4Protocol::m_rttTypeId),
0072:     MakeTypeIdChecker ());
0073: AddAttribute ("SocketType",
0074:     "Socket type of TCP objects.",
0075:     TcpNewReno::GetTypeId ()),
0076:     MakeTypeIdAccessor (&TcpL4Protocol::m_socketTypeId),
0077:     MakeTypeIdChecker ());

0178: Ptr<Socket>
0179: TcpL4Protocol::CreateSocket (TypeId socketTypeId)
0180: {
0181:   NS_LOG_FUNCTION_NOARGS ();
0182:   ObjectFactory rttFactory;
0183:   ObjectFactory socketFactory;
0184:   rttFactory.SetTypeId (m_rttTypeId);
0185:   socketFactory.SetTypeId (socketTypeId);
0186:   Ptr<RttEstimator> rtt = rttFactory.Create<RttEstimator> ();
0187:   Ptr<TcpSocketBase> socket = socketFactory.Create<TcpSocketBase> ();
0188:   socket->SetNode (m_node);
0189:   socket->SetTcp (tls);
0190:   socket->SetRtt (rtt);
0191:   m_sockets.push_back (socket);
0192: return socket;
```
Determine and obtain the congestion window size (3)

- Then, TcpNewReno::window() is called in TcpSocketBase::SendPandingData()

```c++
00102:   uint32_t
00103:   TcpNewReno::Window (void)
00104:   {
00105:     NS_LOG_FUNCTION (this);
00106:     return std::min (m_rWnd.Get (), m_cWnd.Get ());
00107:   }
```

- All TCP variants have their own NewACK(), DupACK()
Procedure of data packet receiving of TCP in n3

- Function calls in TcpSocketBase

- Receive callback

  - ForwardUp()
    - DoForwardUp()
      - ProcessEstablished()
        - ReceivedData()
          - Sink application

  - IP layer

  - Transport layer

  - TcpSocketBase
    - Check the state of TCP
      - Read TCP flags in header
        - Transmit ACK packet
          - Insert the data packet to receive buffer

  - Application layer
FowardUP() \rightarrow ProcessEstablished()

- Read TCP option
- Check the state of TCP & Call proper Action function

```c
00990:     switch (m_state)
00991:     {
00992:         case ESTABLISHED:
00993:             ProcessEstablished (packet, tcpHeader);
00994:             break;
00995:         case LISTEN:
00996:             ProcessListen (packet, tcpHeader, fromAddress, toAddress);
00997:             break;
00998:         case TIME_WAIT:
00999:             // Do nothing
01000:                 break;
01001:         case CLOSED:
```
ProcessEstablished() → ReceivedData()

- Read TCP flags in header
- Call the proper function according to packet type

```c
void TcSocketBase::ProcessEstablished (Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
    NS_LOG_FUNCTION (this << tcpHeader);

    // Extract the flags. PSH and URG are not honoured.
    uint8_t tcpflags = tcpHeader.GetFlags () & ~(TcpHeader::PSH | TcpHeader::URG);

    // Different flags are different events
    if (tcpflags == TcpHeader::ACK)
    {
        ReceivedAck (packet, tcpHeader);
    }
    else if (tcpflags == TcpHeader::SYN)
    {
        // Received SYN, old NS-3 behaviour is to set state to SYN_RCVD and
        // respond with a SYN+ACK. But it is not a legal state transition as of
        // RFC793. Thus this is ignored.
    }
    else if (tcpflags == (TcpHeader::SYN | TcpHeader::ACK))
    {
        // No action for received SYN+ACK, it is probably a duplicated packet
    }
    else if (tcpflags == TcpHeader::FIN || tcpflags == (TcpHeader::FIN | TcpHeader::ACK))
    {
        // Received FIN or FIN+ACK, bring down this socket nicely
        PeerClose (packet, tcpHeader);
    }
    else if (tcpflags == 0)
    {
        // No flags means there is only data
        ReceivedData (packet, tcpHeader);
        if (m_rxBuffer->finished ())
        {
            PeerClose (packet, tcpHeader);
        }
    }
}```
ReceivedData() → Rx_buffer

- Transmit ACK packet
- Insert the data packet to receive buffer

```cpp
02169: void
02170: TcpSocketBase::ReceivedData (Ptr<Packet> p, const TcpHeader& tcpHeader)
02171: {
02172:     NS_LOG_FUNCTION (this << tcpHeader);
02173:     NS_LOG_LOGIC ("seq " << tcpHeader.GetSequenceNumber () <<
02174:                     " ack " << tcpHeader.GetAckNumber () <<
02175:                     " pkt size " << p->GetSize () );
02176:     // Put into Rx buffer
02177:     SequenceNumber32 expectedSeq = m_rxBuffer->NextRxSequence ();
02178:     if (!m_rxBuffer->Add (p, tcpHeader))
02179:     {
02180:         // Insert failed: No data or RX buffer full
02181:         SendEmptyPacket (TcpHeader::ACK);
02182:         return;
02183:     }
```
Summary: TCP Packet Processing in NS-3

1. Send ()

   STM of TCP
   States_t=ESTABLISHED
   Tx_buffer

   2. SendPendingData()

   3. SendDataPacket

   4. SendPacket()

   TcpL4Protocol

   Application

   IP layer

   DoForwardUp()

   ForwardUp()

   ReceivedData()

   ProcessEstablished()

   Receive callback

   Receive callback

   TcpSocketBase

   IP layer

   Sink application

   Transport layer

   Application
**Example: Auto-tuning**

- **rwnd has only 16bit field in TCP header**
  - Maximum available window size is $2^{16}=65535$ byte (46 packets with max segment size)
  - if RTT=50ms, throughput = $65535 \times 8 / 0.05 = 10$Mbps

- **Current long fat network needs larger rwnd value**
Practice 1: Auto-tuning using window scale option

- rwnd has only 16bit field (i.e., “window”) in TCP header
  - Maximum available window size is $2^{16}=65565$ byte (46 packets with max segment size)
  - if RTT=50ms, throughput = $65565 \times 8/0.05 = 10$Mbps

- Window scaling option
  - Increase rwnd field from 64 KBytes to 1 GBytes using a scaling factor
  - Scaling factor (8bit) : 0~14
  - If scaling factor is 5 and rwnd in TCP header is 65565, actual rwnd value is $64k \times 2^5=2Mbyte$

- Auto-tuning
  - linux default scheme (at the receiver)
  - $rwnd = \min\{2 \times w_s, \ tcp \ rmem \ max\}$
    - $w_s$ is the largest transmission window observed by the receiver so far
    - $tcp \ rmem \ max$ is a settable parameter
1. Create the TCP-option field (1/2)

- TcpOption class

```
enum Kind {
    // Remember to extend IsKindKnown() with new value, when adding values here
    END = 0,      // !< END
    NOP = 1,      // !< NOP
    MSS = 2,      // !< MSS
    WINSCALE = 3, // !< WINSCALE
    TS = 8,       // !< TS
    UNKNOWN = 255 // !< not a standardized value; for unknown recv'd options
};
```

- Tcp-option.cc/h
Related Function Calls

1. Send ()
   
   STM of TCP
   
   States_t=ESTABLISHED
   
   1. Add TCP option (sender)
   
   2. SendPendingData()
   
   3. SendDataPacket()
   
   4. SendPacket()

2. Read TCP option (receiver)

3. SendPendingData()

4. SendPacket()

1. Add TCP option (receiver)

ReceivedData()

ProcessEstablished()

DoForwardUp()

ForwardUp()

Receive callback
1. Create the TCP-option field (2/2)

- Tcp-option-winscale.cc/h

```cpp
class TcpOptionWinScale : public TcpOption
{
public:
    TcpOptionWinScale ();
    virtual ~TcpOptionWinScale ();

    static TypeId GetTypeId (void);
    virtual TypeId GetInstanceTypeId (void) const;

    virtual void Print (std::ostream &os) const;
    virtual void Serialize (Buffer::Iterator start) const;
    virtual uint32_t Deserialize (Buffer::Iterator start);

    virtual uint8_t GetKind (void) const;
    virtual uint32_t GetSerializedSize (void) const;

    uint8_t GetScale (void) const;
    void SetScale (uint8_t scale);

protected:
    uint8_t m_scale; // Window scaling in number of bit shift
};
```

Call in tcp-socket-base.cc
2. Add option

```cpp
void
TcpSocketBase::AddOptions (TcpHeader & header)
{
    NS_LOG_FUNCTION (this << header);

    // The window scaling option is set only on SYN packets
    if (m_winScalingEnabled && (header.GetFlags () & TcpHeader::SYN))
    {
        AddOptionWScale (header);
    }

    if (m_timestampEnabled)
    {
        AddOptionTimestamp (header);
    }
}

void
TcpSocketBase::AddOptionWScale (TcpHeader & header)
{
    NS_LOG_FUNCTION (this << header);
    NS_ASSERT (header.GetFlags () & TcpHeader::SYN);

    Ptr<TcpOptionWinScale> option = CreateObject<TcpOptionWinScale> () ;

    // In naming, we do the contrary of RFC 1323. The sended scaling factor
    // is Snd.Wind.Scale (and not Rcv.Wind.Scale)
    m_sndScaleFactor = CalculateWScale () ;
    option->SetScale (m_sndScaleFactor);

    header.AppendOption (option);

    NS_LOG_INFO (m_node->GetId () << " Send a scaling factor of " <<
                static_cast<int> (m_sndScaleFactor));
```
3. Read option

- In DoForwardUp()

```cpp
void TcpSocketBase::ReadOptions (const TcpHeader& header)
{
    NS_LOG_FUNCTION (this << header);
    if ((header.GetFlags () & TcpHeader::SYN))
    {
        if (m_winScalingEnabled)
            m_winScalingEnabled = false;
        if (header.HasOption (TcpOption::WINScale))
            m_winScalingEnabled = true;
        ProcessOptionWScale (header.GetOption (TcpOption::WINScale));
        ScaleSsThresh (m_sndScaleFactor);
    }
}
```
3. Estimate the cwnd

- Receiver has RTT values but not cwnd
- Using RTT trace, estimate the cwnd

```cpp
/// cwnd estimation
if(StartCwndEst && (header.GetFlags() & TcpHeader::ACK)){
    NS_LOG_INFO(" m_cwnd_est_temp : " << (int)m_cwnd_est_temp << " m_cwnd_est : " << (int)m_cwnd_est);
    if(m_time_for_cwnd_est + m_lastRtt_temp > (Time)Simulator::Now().GetNanoSeconds ()){
        m_cwnd_est_temp++;
    } else{
        m_cwnd_est = m_cwnd_est_temp;
        //NS_LOG_INFO("m_cwnd_Est : " << (int)m_cwnd_est_temp);
        m_cwnd_est_temp = 0;
        m_time_for_cwnd_est = (Time)Simulator::Now().GetNanoSeconds ();
    }
}
```

→ counting incoming packet during measured RTT
4. Determine scale factor

Determine window size as double of cwnd

Calculating scale factor

- It is called instead of calculateWScale().
Practice 1: Auto-tuning using window scale option

- **ns_log_function trace**
  - Scratch/tcp-large-transfer.cc
    → line 74: LogComponentEnable("TcpSocketBase", LOG_WARN);

- **./waf --run scratch/tcp-large-transfer**

- **Then, where should the functions be called??**
  - Addoptions (tcpheader)
    - SendDataPacket() – line 1974
    - SendEmptyPacket() – line 1700
  - Readoptions(tcpheader)
    - DoForwardUp() – line 912, 1018
  - DetermineScale(header,~~~)
    - SendEmptyPacket() – line 1697
Multi-path TCP (MPTCP)

- Multi-path TCP
  - Provide concurrent & opportunistic transmission on multiple path in transport layer
  - Provide seamless vertical H/O by addition/elimination of subflows

→ Increase the resilience of connectivity
→ Increase the efficiency of resource usage
→ Backwards-compatible with regular TCP and network
Research and Standardization of MPTCP

• **Research for MPTCP**

• **Standardization of MPTCP**
  - Current status : Active WG in IETF
  - Published draft
    - RFC 6897 – API of MPTCP
    - RFC 6182 – Architecture of MPTCP
    - RFC 6356 – Congestion control
    - RFC 6824 – Multi-path operation with multiple address
    - RFC 6181 - Security
Who is using MPTCP?

- **Linux kernel implementation**
  - IP networking Lab in UCL, Belgium
  - Provide stable linux kernel of MPTCP

- **Apple iOS7**
  - Improve resilience for Siri & software update server

- **KT**
  - GiGA LTE : Improve throughput using WiFi and LTE-A
GIGA LTE

- First commercialized MPTCP by KT and Samsung
- It might be through the proxy server in PDN gateway
- It supports all kinds of application except for mVoIP
Architecture of MPTCP (1)

- Architecture MPTCP server and receiver
Architecture of MPTCP (2)

- Usage scenario of MPTCP

Regular server and MPTCP proxy server
Architecture of MPTCP (3)

- **TCP header**

<table>
<thead>
<tr>
<th>Source port address (16 bits)</th>
<th>Destination port address (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number (32 bits)</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment number (32 bits)</td>
<td></td>
</tr>
<tr>
<td>Header Length (4 bits)</td>
<td>Reserved (4 bits)</td>
</tr>
<tr>
<td>Flags (8 bits)</td>
<td>Window (16 bits)</td>
</tr>
<tr>
<td>Checksum (16 bits)</td>
<td>Urgent pointer (16 bits)</td>
</tr>
<tr>
<td>Options and Padding (0-320 bits)</td>
<td></td>
</tr>
</tbody>
</table>

- **MPTCP in TCP option field**
  - Data sequence number (32bit), data length (16bit)
  - Handed as other TCP option examples (window scale, timestamp, selective ACK, etc)
  - Sequence number in TCP → subflow sequence number
  - Data sequence number in Option field → global sequence number
Retransmission & Reordering in MPTCP

- **Reordering in MPTCP**
  - Subflow level reordering
    - ACK/ Dupack is transmitted in subflow level
    - Caused by packet loss in networks
      → Dupack or RTO in subflow level
  - Global reordering
    - Caused by difference of subflow’s one-way delay
Congestion control in MPTCP

• **Uncoupled TCP Reno**
  • The simplest algorithm that one can imagine is to use TCP New Reno over each of the subflow
  
  - For each ACK on flow $i$: $w_i = w_i + \frac{1}{w_i}$
  - For each loss on flow $i$: $w_i = \frac{w_i}{2}$

• **Fully Coupled**
  • For TCP fairness with single TCP
  • Better congestion balancing than reno
  
  - For each ACK on flow $i$: $w_i = w_i + \frac{1}{w}$
  - For each loss on flow $i$: $w_i = \frac{w_i}{2}$

• **Linked Increases**
  • Compensating the difference of RTTs
  • Guarantee the throughput improvement and TCP-fairness
  • Standardized in IETF

\[
\begin{align*}
  w_i &= w_i + \min\left(\frac{a}{w}, \frac{1}{w_i}\right) \\
  w_i &= \frac{w_i}{2}
\end{align*}
\]

\[
a = \hat{\omega}\left(\frac{\max_r \hat{\omega}_r/RTT_r^2}{\sum_{r \in R} \hat{\omega}_r / RTT_r^2}\right)
\]
Ns3 module for MPTCP

- Submitted by: Bachir CHIHANI
- Defect of contributed code
  - Compatible with 3.6~3.10
  - Bugs in ACK transmission and receiver architecture.

- Provided MPTCP is modified to be operated in recent ns3 (~3.10)
  - Need additional modified to provide attribute value
Architecture of MPTCP in ns-3

- **MpTcpSocketBase.cc/h**
  - Implementation of additional MPTCP operation
  - Add/terminate MPTCP subflow, congestion control, reordering in receiver ...
MPTCP header

- They are added on TCP Option field
  - Connection setup
  - Add/remove address

```c
class OptDataSeqMapping : public TcpOptions {
public:
    virtual ~OptDataSeqMapping();
    uint64_t dataSeqNumber;
    uint16_t dataLevelLength;
    uint32_t subflowSeqNumber;
    OptDataSeqMapping(TcpOption_t oName, uint64_t dSeqNum, uint16_t dLevelLength, uint32_t sfSeqNum);
};
```
MPTCP connection establishment

**src : Application**
: Connect()

**src : MPTCP**
: SYN, OPT_MPC
: SYN, ACK, OPT_MPC
: ACK
: Connection established

**dst : MPTCP**
: Listen()

**dst : Application**
: Connection established

**src @i : Subflow**
: ACK, OPT_ADDR @src1
: ACK, OPT_ADDR @dst2

**dst @i : Subflow**
: SYN, OPT_JOIN
: SYN, ACK
: ACK
Procedure of packet sending of TCP (Revisit)

1. Send(): check the state of TCP

   STM of TCP
   - States_t=ESTABLISHED
   - Tx_buffer

   2. SendPendingData(): Check the window size and tx buffer
   3. SendDataPacket(): Add TCP header and Option
   4. SendPacket(): tx packet to l3layer

application

scenario file

transport layer

ip layer

TCP L4 Protocol

TcpSocketBase
Procedure of packet sending of MPTCP in n3

1. FillBuffer(): check the state of MPTCP
2. SendBufferedData(): call ProcessAction()

STM of TCP

States_t=ESTABLISHED

3. SendPendingData()
   1) Select the subflow (scheduler+CC)
   2) Add TCP header MPTCP header

4. SendPacket(): tx packet to l3layer

Application

Scenario file

Transport layer

Ip layer
Application : Fill the buffer

- **scratch/mptcp_simple_scen.cc**

```cpp
void WriteUntilBufferFull (Ptr<Socket> localSocket, unsigned int txSpace)
{
    while (currentTxBytes < totalTxBytes && localSocket->GetTxAvailable () > 0)
    {
        uint32_t left = totalTxBytes - currentTxBytes;
        uint32_t toWrite = std::min(writeSize, localSocket->GetTxAvailable ());
        toWrite = std::min( toWrite, left);
        int amountBuffered = localSocket->FillBuffer (&data[currentTxBytes], toWrite);
        currentTxBytes += amountBuffered;
    }
    // variateDelay(client);
    localSocket->SendBufferedData();
}
```

- **src/internet/model/mp-tcp-socket-base.cc**

```cpp
bool MpTcpSocketImpl::SendBufferedData ()
{
    //NS_LOG_FUNCTION_NOARGS();
    uint8_t sFlowIdx = lastUsesdFlowIdx; // I prefer getSubflowToUse (), but this one gives the next one
    Ptr<Ipv4L3Protocol> ipv4 = m_node->GetObject<Ipv4L3Protocol> ();
    if (!ProcessAction (sFlowIdx, ProcessEvent (sFlowIdx, APP_SEND) ))
    {
        return false; // Failed, return zero
    }
    return true;
}
```
Subflow selection (scheduler + congestion control algorithm)

```cpp
bool MpTcpSocketImpl::SendPendingData ()
{
    NS_LOG_INFO("---------------------------------------------");
    //NS_LOG_FUNCTION_NOARGS ();
    
    while ( count < subflows.size () )
    {
        count ++;
        window = std::min (AvailableWindow (lastUsedsFlowIdx), sendingBuffer->PendingData ());
        if ( window == 0 || window < subflows[lastUsedsFlowIdx]->MSS) //added by PSY for R-TSC
        {
            // No more available window in the current subflow, try with another one
            lastUsedsFlowIdx = getSubflowToUse ();
        }
        else
        {
            NS_LOG_LOGIC ("MpTcpSocketImpl::SendPendingDat
            break;
        }
    }

    uint8_t
    MpTcpSocketImpl::getSubflowToUse ()
    {
        uint8_t nextSubFlow = 0;
        switch ( distribAlgo )
        {
            case Round_Robin :
                nextSubFlow = (lastUsedsFlowIdx + 1) % subflows.size ();
                break;
            default:
                break;
        }
        return nextSubFlow;
    }
```

Check the window size of subflow and select the other subflow

- **Subflow Scheduling**
  - Not defined in standard
  - Simply, use Round Robin
Add TCP/MPTCP header

```cpp
bool MpTcpSocketImpl::SendPendingData ()
{
    NS_LOG_INFO("=================================================================");
    //NS_LOG_FUNCTION_NOARGS ();

    MpTcpHeader header;
    header.SetSourcePort (sPort);
    header.SetDestinationPort (dPort);
    header.SetFlags (flags);
    header.SetSequenceNumber (sFlow->TxSeqNumber);
    header.SetAckNumber (sFlow->RxSeqNumber);
    header.SetWindowSize (AdvertisedWindowSize ());
    // save the seq number of the sent data
    sFlow->AddDSNMapping (lastUsedsFlowIdx, nextTxSequence, siz
    header.AddOptDSN (OPT_DSN, nextTxSequence, size, sFlow--

    //
    m_mptcp->SendPacket (pkt, header, sAddr, dAddr);
}
```
Congestion control algorithm in MPTCP

- Congestion control algorithm
  - Uncoupled
  - Fully coupled
  - Linked Increases Algorithm (std)

```c
uint32_t
MptcpSocketImpl::AvailableWindow (uint8_t sFlowIdx)
{
    //NS_LOG_FUNCTION_NOARGS ();
    MptcpSubFlow * sFlow = subflows[sFlowIdx];
    uint32_t window = std::min (static_cast<uint32_t> (remoteRecvWnd),
                                static_cast<uint32_t> (sFlow->cwnd)) * sFlow->MSS;

    uint32_t unAcked = sFlow->maxSeqNb - sFlow->highestAck;

    if (window < unAcked) //DataCount
    {
        NS_LOG_LOGIC("MptcpSocketImpl::AvailableWindow -> Available Tx window is 0");
        return 0; // No space available
    }
    else
    {
        return (window - unAcked);//DataCount); // Amount of window space available
    }
}```
Practice 2 : Comparison of congestion control algorithm of MPTCP (1)

• According to CC, increment and decrement of window are different
• We should implement OpenCWND(), reduceCWND()
• OpenCWND() : line 2323~2344
  • when it receives new ACK, fill the blank

    ```c
    switch ( AlgoCC )
    {
    case Linked_Increases:
        // add your code
        break;
    case Uncoupled_TCPs:
        // add your code
        break;
    case Fully_Coupled :
        // add your code
        break;
    default :
        break;
    }
    }
    ```

• Hint : use calculate_alpha()
Reminder: Congestion control in MPTCP

- **Uncoupled TCP Reno**
  - The simplest algorithm that one can imagine is to use TCP New Reno over each of the subflow

- **Fully Coupled**
  - For TCP fairness with single TCP
  - Better congestion balancing than reno

- **Linked Increases**
  - Compensating the difference of RTTs
  - Guarantee the throughput improvement and TCP-fairness
  - Standardized in IETF

\[
\begin{align*}
  w_i &= w_i + \frac{1}{w_i} \\
  w_i &= \frac{w_i}{2}
\end{align*}
\]

\[
\begin{align*}
  w_i &= w_i + \frac{1}{w} \\
  w_i &= \frac{w_i}{2}
\end{align*}
\]

\[
\begin{align*}
  w_i &= w_i + \min\left(\frac{a}{w}, \frac{1}{w_i}\right) \\
  w_i &= \frac{w_i}{2}
\end{align*}
\]

\[
a = \hat{\omega}\left(\frac{\max_r \hat{\omega}_r / RTT_r^2}{\sum_{r \in R} \hat{\omega}_r / RTT_r^2}\right)
\]
Practice 2: Comparison of congestion control algorithm of MPTCP (1)

- According to CC, increment and decrement of window are different
- We should implement OpenCWND(), reduceCWND()
- OpenCWND() – when it receives new ACK, fill the blank

```cpp
    case Linked_Increases:
      calculate_alpha();
      increment = std::min( alpha / totalCwnd, 1.0 / cwnd );
      NS_LOG_ERROR ("Subflow "<<(int)sFlowIdx" Congestion Control (Linked_Increases):
      break;

    case Uncoupled_TCPs:
      increment = 1.0 / cwnd;
      NS_LOG_ERROR ("Subflow "<<(int)sFlowIdx" Congestion Control (Uncoupled_TCPs) in
      break;

    case Fully_Coupled:
      increment = 1.0 / totalCwnd;
      NS_LOG_ERROR ("Subflow "<<(int)sFlowIdx" Congestion Control (Fully_Coupled) inc
      break;
    default:
      break;
  }
```
Practice 2: Comparison of congestion control algorithm of MPTCP (2)

- ReduceCWND() - when receive duplicated ACK

```c
void MpTcpSocketImpl::reduceCWND (uint8_t sFlowIdx)
{
    MpTcpSubFlow *sFlow = subflows[sFlowIdx];
    double cwnd = sFlow->cwnd;
    calculateTotalCWND ();

    sFlow->ssthresh = (std::min(static_cast<uint32_t>(remoteRecvWnd), static_cast<uint32_t>(sFlow->cwnd))) / 2; // Per RFC2581
    sFlow->ssthresh = std::max (sFlow->ssthresh, 2 * sFlow->MSS);

    double qThroughput = getGlobalThroughput();
    uint64_t 1Delay = getPathDelay(sFlowIdx);

    switch ( AlgoCC )
    {
    case Uncoupled_TCPs:
        sFlow->cwnd = std::max (cwnd / 2, 1.0);
        NS_LOG_INFO (Simulator::Now().GetSeconds() << " MtcpSocketImpl -> " << " localToken " << " localToken" << " Subflow " << (sFlow->cwnd));
        break;
    case Linked_Increases:
        sFlow->cwnd = std::max (cwnd / 2, 1.0);
        NS_LOG_INFO (Simulator::Now().GetSeconds() << " MtcpSocketImpl -> " << " localToken " << " localToken" << " Subflow " << (sFlow->cwnd));
        break;
    case Fully_Coupled:
        sFlow->cwnd = std::max (cwnd - totalCwnd / 2, 1.0);
        NS_LOG_INFO (Simulator::Now().GetSeconds() << " MtcpSocketImpl -> " << " localToken " << " localToken" << " Subflow " << (sFlow->cwnd));
        break;
    default:
        sFlow->cwnd = 1;
        NS_LOG_INFO (Simulator::Now().GetSeconds() << " MtcpSocketImpl -> " << " localToken " << " localToken" << " Subflow " << (sFlow->cwnd));
        break;
    }

    sFlow->phase = Congestion_Avoidance;
    // sFlow->reseqnumber = sFlow->highestAck + 1; // Start from highest Ack
    sFlow->rtt->IncreaseMultiplier (); // DoubleValue timeout value for next retx timer
}
```
Practice 2 : Comparison of congestion control algorithm of MPTCP (3)

- Mptcp_simple_scen.cc

```c
int main(int argc, char *argv[])
{
    bool verbose;
    CongestionCtrl_t cc = Fully_Coupled;
    PacketReorder_t pr = D_SACK;
    int arg1 = -1, arg2 = -1, arg3 = -1, arg4 = -1;
    int sf = 2; // number of subflows
    CommandLine cmd;
    cmd.AddValue("verbose", "Tell application to log if true", verbose);
    cmd.AddValue("level", "Tell application which log level to use:
           \n           \n           t - 0 = ERROR
           t - 1 = WARN
           t - 2 = DEBUG
           t - 3 = INFORMAL
           t - 4 = DEBUG镒"");
    cmd.AddValue("cc", "Tell application which congestion control algorithm to use:
           \n           t - 0 = Uncoupled TCPs
           t - 1 = Linked Increase
           t - 2 = Eifel
           t - 3 = INFORMAL
           t - 4 = DEBUG镒"");
    cmd.AddValue("pr", "Tell application which packet reordering algorithm to use:
           \n           t - 0 = NoPR_Algo
           t - 1 = Eifel
           t - 2 = Eifel
           t - 3 = INFORMAL
           t - 4 = DEBUG镒"");
    cmd.AddValue("sf", "Tell application the number of subflows to be established between endpoints", arg4);
    cmd.Parse (argc, argv);

    lSocket->SetCongestionCtrlAlgo (cc);
    lSocket->SetDataDistribAlgo (Round Robin);
    lSocket->SetPacketReorderAlgo (pr);
    lSocket->Bind ();
}
Procedure of data packet receiving of TCP (Revisit)

- Function calls in TcpSocketBase

- IP layer
  - Receive callback

- Transport layer
  - ForwardUp()
  - DoForwardUp()
  - ProcessEstablished()
  - ReceivedData()

- TcpSocketBase
  - Check the state of TCP
  - Read TCP flags in header
  - Transmit ACK packet
  - Insert the data packet to receive buffer

- Application layer
  - Sink application
Architecture of MPTCP receiver (1)

- Procedure of packet receiving in MPTCP

Receive callback

ForwardUp()

Check the state of TCP

TcpSocketBase

If (opt->optName == OPT_DSN)

1. Check the state of MPTCP
2. Read TCP flags in header
3. Transmit ACK packet
4. Insert the data packet to receive buffer

Sink application
Implementation of hierarchical receive buffer

1. **Subflow sequence mapping**
   1. Unordered case
      - Packet loss in subflow
      → Transmission duplicated ACK
      → store the packet in subflow-level receive buffer
   2. Ordered case
      → Go to Global sequence mapping

2. **Global sequence mapping**
   1. Unordered case
      - Wait for ordered packets on other subflow in global reordering buffer
   2. Ordered case
      - Forward up to application layer
Practice 3: Implementation of hierarchical receive buffer

```c
} else if (opt->optName == OPT_DSN)
{
    // data sequence or stream it
    OptDataSeqMapping * optDSN = (OptDataSeqMapping *) opt;
    TxACK = true;
    *dataLen = optDSN->dataLevelLength;
    Ptr<Packet> packet = pkt;
    uint32_t pktLen = *dataLen;
    if (optDSN->subflowSeqNumber == sFlow->RxSeqNumber)
    {
    // Subflow sequence mapping

1. Transmit ACK

    if (optDSN->dataSeqNumber == nextRxSequence)
    {
    // Global sequence mapping

2. Read the buffers and Flush it

    else if (optDSN->dataSeqNumber > nextRxSequence)
    {
    // Global sequence mapping

1. Store global receive buffer

    else
    {
        //action = NO_ACT;
    }
    } else if ((optDSN->subflowSeqNumber) > sFlow->RxSeqNumber)
    {
    // Packet loss case

1. Store subflow receive buffer

```
Running: Mptcp_simple_scen.cc (1)

- Modified from example/tcp/tcp-large-transfer.cc

Point-to-point link: 5Mbps, 100ms

Point-to-point link: 5Mbps, 100ms
Running : Mptcp_simple_scen.cc (2)

• **Input command**

```c
CommandLine cmd;
cmd.AddValue("verbose", "Tell application to log if true", verbose);
cmd.AddValue("level", "Tell application which log level to use:\n \t 0 = ERROR \n \t 1 = WARN \n \t 2 = INFO \n \t 3 = DEBUG\n", level);
```

```c
cmd.AddValue("cc", "Tell application which congestion control algorithm to use:\n \t 0 = Uncoupled_TCPs\n", cc);
```

```c
cmd.AddValue("sf", "Tell application the number of subflows to be established between endpoints", arg4);
```

```c
cmd.Parse (argc, argv);
```

```c
cc = (arg1==-1 ? Fully_Coupled:(CongestionCtrl_t) arg1);
sf = (arg4 = -1 ? 2 : arg4);
```

• **Deploy Node and MPTCP stack**

```c
NodeContainer nodes;
nodes.Create(2);
client = nodes.Get(0);
server = nodes.Get(1);
```

```c
MpInternetStackHelper stack;
stack.Install(nodes);
```
Running : Mptcp_simple_scen.cc (3)

- **Link setup**

```
vector<IPv4InterfaceContainer> ipv4Ints;
for(int i=0; i < sf; i++)
{
    // Creation of the point to point link between hots
    PointToPointHelper p2pLink;
p2pLink.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
p2pLink.SetChannelAttribute("Delay", StringValue("100ms"));

    NetDeviceContainer netDevices;
etDevices = p2pLink.Install(nodes);

    // Attribution of the IP addresses
    std::stringstream netAddr;
etAddr << "10.1." << (i+1) << ".0";
    string str = netAddr.str();

    Ipv4AddressHelper ipv4addr;
ipv4addr.SetBase(str.c_str(), "255.255.255.0");
ipv4InterfaceContainer interface = ipv4addr.Assign(netDevices);
ipv4Ints.insert(ipv4Ints.end(), interface);
}
```

- **Configure MPTCP application**

```
// Configuration of the Client/Server application
uint32_t servPort = 5000;
NS_LOG_INFO("address " << ipv4Ints[0].GetAddress(1));
ObjectFactory m_sf;
m_sf.SetTypeId("ns3::MtpcpSocketSink");
m_sf.Set("Protocol", StringValue("ns3::TcpSocketFactory");
m_sf.Set("Local", AddressValue(InetSocketAddress (ipv4Ints[0].GetAddress(1),
    Ptr<Application> sapp = m_sf.Create<Application> () ;
server->AddApplication(sapp);
ApplicationContainer Apps;
Apps.Add(sapp);

//ApplicationContainer serverApps = sink.Install(server);
Apps.Start(Seconds(0.0));
Apps.Stop(Seconds(simDuration));
```
• MPTCP parameter setting

```cpp
lSocket = new MpTcpSocketImpl (client);
/*
 */
lSocket->SetCongestionCtrlAlgo (cc);
lSocket->SetDataDistribAlgo (Round_Robin);
lSocket->Bind ();
```

• void StartFlow(Ptr<MpTcpSocketImpl> localSocket, Ipv4Address servAddress, uint16_t servPort)
  • Establish MPTCP connection
  • Set the number of subflow and send/receive buffer size

• void WriteUntilBufferFull (Ptr<Socket> localSocket, unsigned int txSpace)
  • Push the data packet to send buffer
Running : Mptcp_simple_scen.cc (5)

- **ns_log_function trace**
  - Scratch/mptcp-simple_scen.cc
    - line 99 : LogComponentEnable("MpTcpSocketBase", LOG_WARN);

- ```
  ./waf --run "scratch/mptcp_simple_scen --cc=1"
  ```

<table>
<thead>
<tr>
<th>Subflow</th>
<th>Congestion Control</th>
<th>Increment (Uncoupled_TCPs)</th>
<th>Cwnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.0180859</td>
<td>55.2916</td>
</tr>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.0179514</td>
<td>55.7061</td>
</tr>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.01808</td>
<td>55.3097</td>
</tr>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.0179456</td>
<td>55.7241</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subflow</th>
<th>Congestion Control</th>
<th>Increment (Linked_Increases)</th>
<th>Alpha</th>
<th>Cwnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.0049422</td>
<td>0.0049422</td>
<td>50.5085</td>
</tr>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.00494196</td>
<td>0.00494196</td>
<td>50.661</td>
</tr>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.00494172</td>
<td>0.00494172</td>
<td>50.5135</td>
</tr>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.00494148</td>
<td>0.00494148</td>
<td>50.6659</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subflow</th>
<th>Congestion Control</th>
<th>Increment (Fully_Coupled)</th>
<th>Cwnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.00989055</td>
<td>50.6647</td>
</tr>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.00988959</td>
<td>50.4418</td>
</tr>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.00988862</td>
<td>50.6746</td>
</tr>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.00988765</td>
<td>50.4517</td>
</tr>
<tr>
<td>1</td>
<td>Congestion Control</td>
<td>Increment is 0.00988668</td>
<td>50.6845</td>
</tr>
<tr>
<td>0</td>
<td>Congestion Control</td>
<td>Increment is 0.00988572</td>
<td>50.4616</td>
</tr>
</tbody>
</table>
Q & A